1. Implementation Overview

This is a C++ program that applies a range of image filters and orthographic projections to 2D input images and 3D data volume. As a summary of our results, we have completed all the 2D image filters (11 in total), both 3D data volume filters, all the 3 orthographic projections, and the slicing functionality. We have also finished the following additional tasks: projection over a thin slab, comments and unit tests for our code, a license, a ReadMe file, a documentation (pdf generated by Doxygen), and compiled Windows and MacOS executables.

2. Code Structure

We have implemented one main function and 6 Classes, their names and functionalities as follows:

Image - represents an image and memory and sets its attributes
Filter - performs all 2D filters on an image
Volume - construct 3D volume data, write processed volume into images, apply 3D filters
Projection - computes 3 orthographic projections on 3D data volume.
Slice - slices 3D volume data
Tests - tests all the functions defined in Filter.cpp, Volume.cpp, Slice.cpp and Projection.cpp
Each class has one header file (.h) and one source file (.cpp)

3. Details of Implementation

3.1 2D Image Filters

The 2D Image filters can be further categorized as follows:

- Color Correction: This included implementing filters that modified images per pixel. The following filters have been implemented-
    1. *Grayscale*: This filter converts a color image to a grayscale or a black and white version by reducing image from 3 RGB channels to 1 RGB channels. The grayscale function calculates the gray values of each pixel in the image as a weighted average of its RGB color channels.
    2. *Automatic Color Balance:* This filter implements the automatic color balance and adjusts each RGB channel so that the average intensity of each channel is equal. The automatic color balance functions, calculates the total of each of the RGB channels and then adjusts their values so that they are all equal.
    3. *Brightness*: This filter is used to adjust the brightness of the image by a user specified value. If the user does not specify a value then the automatic brightness adjustment algorithm is applied. However, if a value for brightness is specified then it is added to each pixel value to modify brightness.
    4. *Histogram Equalization*: This filter is applied to improve the contrast and brightness of the images. The histogram of the image is calculated, followed by evaluating the probability density function, cumulative density function and mapping the resulting values to create a new image.

- Image Blur: In this section the implemented filters blur the image using convolution filters of different kernel size. Following are the filters that have been implemented:
    1. *Median Blur* : To implement this filter, each pixel value of the given image was replaced by the median of those around it and then stored in a new data structure of the size (kernel * kernel * channels). This data is used to create the new image.

2. *Box Blur* : This filter blurs the image by replacing each pixel value by the average of those around it and then stored in a new data structure of the size (kernel * kernel * channels). This data is used to create the new image.
3. *Gaussian Blur* : For this filter operation we need to be given a value of sigma and a value of filter size, then we can calculate the corresponding Gaussian kernel and we need to normalize the Gaussian kernel. We then do a Gaussian process for each pixel point in the image in turn.

- Edge Detection: In this section, edge detection filters are implemented on grayscale images obtained from the grayscale function. Following are the filters that have been implemented:
    1. *Sobel* : This filter uses the sobel edge detection algorithm to compute the gradient of image intensity at each pixel. It uses two 3x3 kernels to calculate horizontal and vertical gradients. The resulting gradients are combined to get the magnitude of the gradient at each pixel, following this a threshold value is applied to obtain the edges.
    2. *Prewitt:* This filter uses the prewitt edge detection algorithm to detect the edges in the image.This algorithm calculates horizontal and vertical gradients. However, the two 3x3 kernels have different values than the ones used in Sobel function. The gradients are then used to get the magnitude which is used to obtain the edges.
    3. *Scharr*: This filter uses the Scharr edge detection algorithm to detect the edges in the image.This algorithm calculates horizontal and vertical gradients. However, the two 3x3 kernels have different values than the ones used in Sobel function. The gradients are then used to get the magnitude which is used to obtain the edges.
    4. *Roberts' Cross filter* uses the Roberts' Cross algorithm, with two 2x2 diagonal kernels, calculating horizontal and vertical gradients respectively. The gradients are then combined to get the magnitude and thus the edges.

3.1.1 The Image Class

Class "Image" represents an image in memory.

The class has a constructor that takes the width, height, channel, and pixel data as arguments and initializes the private members with these values. The class also has a destructor that deallocates any dynamically allocated memory.The public interface of the class includes more functions.

The get_width(), get_height(), get_channel(), get_data() are all accessor functions that return the private members representing the width, height, channel and data of the image respectively. Moreover, the class contains functions such as set_width(), set_height(), set_channel(), set_data() which are used to set the private members. In addition to this, the function padding(), takes the pixel data as an argument, adds padding of 0's around the image, and returns the padded pixel data as a 1D array.

3.1.2 The Filter Class

Class "Filter" provides various image processing methods for color correction, image blur, and edge detection. This class creates an object of the Image class and initializes the object with the input image. The data received from the input image is stored in an unsigned char * and integers representing the width, height and channels of the image in pixels. After the

image object has been initialized, the 2D image filters mentioned in Section 3.1, including all color correction, image blur and edge detection methods, are applied to that image and all the outputs are stored in the output directory.
Overall, this class provides a set of commonly used image processing techniques for color correction, blur, and edge detection in a modular and extensible way.

3.2 3D Functionalities

This part mainly implements two 3D filters: 3D Gussian and 3D median. Then the images are projected orthogonally, they can be further categorized as follows.

3.2.1. Two 3D filters: 3D median and 3D Gussian

These two filters are almost identical to 2D median blur and 2D Gaussian blur, the only difference is that for the 3D filter, it is expected that the image has many channels, in the given test Scans data, for example: the fracture image data has 1300 grayscale images, then it should be considered together as a 3D image with length*width*number of channels at this point with 1300 channels. The 3D filter should be able to process these 1300 grayscale images at once. So at the code level we should iterate through the number of channels again on top of the 2D one and do the calculation in turn.

3.2.2. The Volume Class

The volume class is to construct 3D volume data and write processed volume into images.
It has protected members: width, height, depth, channel, img_size, total_size, outdir(a vector containing string of filename), and data(a unique_ptr to an unsigned char array that holds the pixel data of the volume as a 1D array). For public members, it is about conversion between images and 3D volume. The most important public function is Volume::Volume(std::string data_dir) and void Volume::writeImages(...).
Within Volume::Volume(std::string data_dir), the process is that first read filenames from a user-defined folder, and sort them, then preallocate the data to hold the volume, finally load and concatenate the sorted images into data.
The purpose of the writeImages() function is to write the processed volume data to multiple PNG image files, with each image file corresponding to a different slice of the 3D volume data.

3.2.3. The Projection Class

The Projection class takes in the directory of 3D data, reads in the data and computes 3 orthographic projections as required - Maximum intensity projection (MIP), Minimum intensity projection (MinIP), and Average intensity projection (AIP). The projections can be calculated both over the entire data volume and over a thin slab specified by the user.
To compute the projections, we use an algorithm that processes one 2D image at a time. We choose doing it this way because it has a lower requirement for memory. We do not need to load all the 3D data into memory all at once, which also means that the Projection class is independent from the Volume class. Specifically, in our algorithm we firstly load in the very first 2D image and use it as our base projection. Then we loop over all the 2D images, each time read one image, compute the min/max with base projection, update the results to base projection, free the loaded image, and go into the next iteration. For the mean projection, we scale every image with 1 / (zmax - zmin), and just add them together. Finally the result is stored in the base projection and written to a file using ‘stbi_write’.

3.2.4 The Slice Class

Slice class aims to slice 3D volume data according to use-defined slice plane and position.
Publicly inherited from the Volume class, this class only owns a few slice related private data members, including string slice_plane; int slice_pos, and unique_ptr<unsigned char[]> slice_pic. Slice_pic points to the slice image we want to get. The private function void store_pic(Volume& v) is to store the slice image. For public functions, especially constructor and set functions, we use throw exceptions to avoid slice_pos out of range.
Within void Slice::get_slice_pic(Volume& v), we slice 3D volume data along user-defined plane and position. Suppose the stack of CT scan is arranged from top(0) to bottom(depth). For example, if slice along y-z plane, first preallocate Slice_pic as big as *height*depth*channel*. Then we set i to be row index of slice image, which corresponds to height of volume, j to be col index, corresponding to depth of volume. Suppose we observe from the right. For slice_pic[i,j], the index for the volume is
*idx = slice_pos + i * width + j * img_size*
After adding channels, it becomes
*idx = (slice_pos + i * width + j * img_size) * channel + ch*
For the x-z slice plane, i corresponds to depth of volume, and j corresponds to width. Suppose we observe from above. For slice_pic[i,j], the index of volume is
*idx = (j + slice_pos * width + (depth - 1 - i) * img_size) * channel + ch*

3.3 Tests

The class “Tests” defines test functions for 2D and 3D filters. For each test function, an image is loaded and a filter is applied on it. For example: When the grayscale filter is being tested, the number of channels, height, width of the resulting image is tested. For a grayscale image, the number of channels should be one, the height and width of the output should be the same as the original image.
Similarly, for other color correction filters, it is checked if the resulting image’s pixels have been modified correctly and lie in the expected range. When testing the blur functions, the output image is being compared against an image that has been correctly modified(depending on the Image Blur in question).
The same process is applied to the edge detection functions as well.
For the Projection class, it is tested, for both 3D data sets, if the output projection has the correct size as input images, and by setting up special cases it is checked if the MIP / MinIP / Average AIP computation is correct.
Note: Since the core implementation of Gaussian and Median blur are the same both on 2D and 3D, so in the testing code we only covered the 2D tests by numerical method. Also we have tested the 3*3 convolution operations also by numerical method, in the code comments it has more details on that!

3.4 Performance

Generally speaking, the larger the size of data is, and the larger the filter is, the longer it takes to run our code. Algorithms that do not use filters (e.g., color correction, projection, slicing) have time complexity O(width * height * channels), while those that involve filters (e.g., image blur, edge detection, 3D filters) have complexity O(width * height * channels * kernel_size).
We tested 3D Median and 3D Gaussian filters. With a 3*3 kernel, the algorithms take around 3 min and 7 seconds respectively, while it takes around 8 min and 1 min respectively if choosing 5*5 kernel.

4. Division of Labor

Xuefei Mi (xm421) -

- Filter class[2D image blur(Median Blur and Gaussian Blur)]
- Slice class
- Volume class

Yuyang Wang (yw22) -

- Image class
- Filter class[2D edge detection(Sobel, Prewitt, Scharr)]
- Volume class
- Code architecture and review

Chaofan Wu (cw522) -

- Volume class[3D Median, 3D Gaussian]
- Test class 2D Gaussian numerical testing and 3*3 convolution numerical testing]

Yi Yang (yy3222)

- Projection class
- Filter class[2D edge detection(Roberts)]

Rubab Atwal (ra922)

- Filter class[2D color correction[Auto Color Balance, Grayscale, Brightness, Histogram Equalisation]
- 2D image blur(Median Blur and Box Blur)]
- Test class